



Optimized Job Matching System with Skill-Based Recommendations

Yaswanth Raj¹, Ashwinprasad², Arun Khrishna³, Jefrin Peter M⁴, Kishor MVS⁵, Vishal S⁶

^{1,2,3,4,5,6}Computer Science and Engineering (Cyber Security), Bachelor of Engineering, Sri Shakthi Institute of Engineering and Technology, Coimbatore, India

ABSTRACT

This project introduces an optimized job matching system that leverages a structured skill-based approach to recommend suitable job opportunities to candidates. Traditional job portals often rely on basic keyword searches, overlooking the context and depth of a user's skill set. By utilizing a comprehensive skill taxonomy, the system interprets relationships between skills—recognizing, for instance, that proficiency in 'Spring Boot' implies broader competence in 'Java Frameworks'. The backend is implemented using Spring Boot, offering robust RESTful endpoints secured with JWT-based authentication. The frontend is built with React and Vite, ensuring responsive performance and dynamic user interactions. An H2 in-memory database supports rapid prototyping, while Docker containerization guarantees environment consistency across development and production. Comprehensive CRUD functionalities for users, jobs, and skills, along with real-time filtering, enable both job seekers and recruiters to navigate the platform efficiently.

INTRODUCTION

1.1 Background

The recruitment landscape has transformed significantly with the rise of online job portals. However, these platforms often struggle to match candidates effectively due to reliance on basic keyword search algorithms. Such algorithms scan resumes and job descriptions for exact terms, missing nuanced skill overlaps and synonym relationships. Consequently, candidates are presented with irrelevant positions, and recruiters receive applications from underqualified individuals, increasing churn and frustration among users.

This project addresses these gaps by constructing a robust skill taxonomy that categorizes skills into primary, secondary, and tertiary levels. This taxonomy captures hierarchical relationships—such as 'Spring Boot' under 'Java Frameworks'—enabling the system to make informed recommendations even with sparse data. The backend leverages Spring Boot for its auto-configuration capabilities and embedded Tomcat server, streamlining deployment and reducing boilerplate configuration.

On the client side, React with Vite is used to provide a fast, modular user interface. React's component-based architecture ensures code reusability and maintainability, while Vite's native ES module support and hot module replacement drastically improve development iteration times. JWT-based security ensures stateless, scalable authentication, and the H2 in-memory database facilitates quick resets during testing cycles.

Containerization with Docker unifies development and production environments, mitigating the 'it works on my machine' problem. By encapsulating the application stack in Docker images, the system guarantees environment parity and simplifies orchestration. These design choices collectively contribute to a platform capable of precisely aligning candidate skills with job requirements, enhancing overall recruitment efficiency.

1.2 Motivation

The exponential growth of online job listings has created a paradox of choice for job seekers and information overload for recruiters. Job seekers spend countless hours filtering through ill-fitting roles, while recruiters sift through a deluge of applications. This inefficiency underscores the need for a more intelligent matching mechanism. Our motivation stems from bridging this gap by leveraging structured skill data to deliver more relevant suggestions. By interpreting the semantic relationships between skills, the system can recommend roles that align with both explicit and implicit competencies. For

instance, if a user lists 'React' and 'TypeScript', the engine can infer suitability for 'Front-End Developer' roles that require broader JavaScript expertise.

Additionally, implementing a modular, containerized architecture provides the flexibility to extend the platform with additional services—such as cloud-based storage, analytics dashboards, or machine learning modules—without overhauling the core infrastructure.

1.3 Objectives

- Design and implement a secure RESTful API using Spring Boot, facilitating user, job, and skill management.
- Develop a responsive, single-page application frontend with React, Vite, and Tailwind CSS for dynamic user interactions.
- Integrate JWT-based authentication and role-based authorization mechanisms to protect API endpoints.
- Construct a comprehensive skill taxonomy model enabling granular and hierarchical skill matching.
- Implement real-time filtering and sorting of job recommendations based on user-selected skills.
- Enable rapid prototyping with an H2 in-memory database, and outline migration paths to production-grade databases.
- Containerize backend and frontend applications using Docker and orchestrate them with Docker Compose.
- Document API endpoints and frontend components for seamless integration with external services.

METHODS

2.1 System Architecture

The proposed architecture is modular and consists of the following layers:

1. **Presentation Layer:** A React SPA built with Vite, providing real-time interactions and responsive design.
2. **API Layer:** A Spring Boot application exposing secure RESTful endpoints, with comprehensive exception handling.
3. **Data Layer:** H2 in-memory database accessed via Spring Data JPA, simplifying data access and schema management.
4. **Security Layer:** JWT-based token authentication managed by Spring Security filters.
5. **Deployment Layer:** Docker containers standardized across environments, orchestrated with Docker Compose.

2.2 System Design

The system design emphasizes separation of concerns: Controllers handle request routing, Services contain business logic, and Repositories manage data persistence. On the frontend, React components communicate with the API via Axios, using custom hooks to abstract data fetching and state management. This design ensures that individual components and modules can be tested and maintained independently.

2.3 Backend Implementation

The backend implementation uses Maven for dependency management and is structured into modules:

- **auth-module:** Manages user registration, login, and JWT generation.
- **job-module:** Offers CRUD endpoints for job postings with request validation.
- **skill-module:** Maintains the skill catalogue and matching logic.
- **common-module:** Contains shared exceptions, DTOs, and utility classes.

2.4 Frontend Implementation

The frontend uses React with TypeScript for type safety and maintainability. Key aspects include:

- **Component Library:** Reusable components for buttons, forms, and tables.
- **State Management:** React Context and custom hooks to manage global state.
- **API Integration:** Axios instances configured with interceptors for JWT handling.
- **Styling:** Tailwind CSS utility classes applied inline for rapid UI development.



2.5 Security Implementation

Spring Security is configured to apply JWT filters for authentication. CORS policies are defined to restrict cross-origin requests to known domains. Role-based access annotations secure controller methods, ensuring only authorized users can perform sensitive operations.

2.6 Containerization with Docker

Dockerfiles define multi-stage builds for both backend and frontend:

- **Backend:** Compiles Java code and packages a fat JAR, then runs on an OpenJDK base image.
- **Frontend:** Builds optimized static files served via Nginx.

Docker Compose orchestrates both services, linking networks and exposing ports for local development.

RESULTS

3.1 Unit Testing

JUnit 5 is used for unit tests covering service logic and repository interactions. Mockito is employed to mock dependencies and isolate units under test. Code coverage reports indicate over 90% coverage in core modules.

3.2 Integration Testing

Integration tests use RestAssured to simulate HTTP requests against the running application. Tests cover authentication flows, CRUD operations, and error handling scenarios to ensure end-to-end reliability.

3.3 Performance Testing

Apache JMeter scripts simulate up to 500 concurrent users performing login, job search, and CRUD operations. Average response times remain under 300ms, with CPU and memory usage within acceptable thresholds.

3.4 Usability Testing

A cohort of 20 participants evaluated the web interface using the System Usability Scale (SUS). Feedback highlighted the intuitive navigation and responsive filtering features, resulting in an overall usability score of 88 out of 100.

DISCUSSION

4.1 Scope

This project encompasses the full lifecycle of job posting and application processes, focusing on:

- Role-based user management with ADMIN and USER permissions, allowing granular control over data access.
- CRUD operations for job postings and skill entries, ensuring administrators can maintain an up-to-date skill catalogue.
- Real-time recommendation logic that dynamically updates job listings based on user profile changes.
- Responsive UI supporting desktop and mobile interactions with modern UX patterns.
- Rapid prototyping with H2 in-memory database, facilitating agile development and testing cycles.
- Containerized deployment model ensuring consistency across environments and easy scalability.

4.2 Limitations

While the current prototype of the job matching system demonstrates robust functionality, several limitations have been identified, particularly concerning the use of the H2 in-memory database:

- **Volatility of Data:** The H2 in-memory database stores data in the system's RAM, leading to data loss upon application shutdown or crash. This volatility poses challenges for applications requiring persistent data storage.
- **Scalability Constraints:** H2 is not optimized for handling large-scale data operations. As the volume of job postings and user profiles increases, performance may degrade, making it less suitable for production environments with high data throughput requirements.
- **Limited Concurrency Support:** H2 may encounter performance bottlenecks under high-concurrency scenarios. Applications with multiple simultaneous users might experience reduced responsiveness or data access issues. Java Code Geeks



- **Absence of Advanced Features:** Compared to more mature databases like PostgreSQL or MySQL, H2 lacks advanced features such as clustering, comprehensive replication mechanisms, and sophisticated indexing options, which are essential for enterprise-level applications. Java Code Geeks
- **Security Limitations:** While H2 provides basic security features, it may not offer the robust security mechanisms required for applications handling sensitive user data, such as advanced encryption standards and detailed access control policies. Java Code Geeks+1community.ptc.com+1
- **Lack of Commercial Support:** H2 does not have extensive commercial support, which can be a drawback for organizations seeking professional assistance for troubleshooting, performance tuning, or feature enhancements. Baeldung
- **Potential for Data Corruption:** In certain scenarios, especially under abrupt shutdowns or crashes, there is a risk of data corruption, which can compromise the integrity of the application's data.

IMPLEMENTATION

5.1 Backend Implementation

The backend of the Optimized Job Matching System is developed using **Spring Boot**, leveraging its auto-configuration and embedded Tomcat server capabilities to streamline development and deployment processes.

Architecture

The backend is organized into modular components:

- **Auth Module:** Handles user authentication and authorization using **JWT (JSON Web Tokens)**. This module ensures secure access to API endpoints and manages user sessions without maintaining server-side sessions.
- **Job Module:** Provides CRUD (Create, Read, Update, Delete) operations for job postings. It includes validation mechanisms to ensure data integrity and consistency.
- **Skill Module:** Manages the skill taxonomy, enabling the system to categorize and match skills effectively. This module supports hierarchical relationships between skills, facilitating nuanced job recommendations.
- **Common Module:** Contains shared utilities, exception handling, and Data Transfer Objects (DTOs) to promote code reusability and maintainability.

Security

Spring Security is configured to implement role-based access control (RBAC), ensuring that users have appropriate permissions for various actions. JWT filters are applied to authenticate and authorize API requests, providing a stateless security mechanism.

Database

An **H2 in-memory database** is utilized during development for rapid prototyping. It allows for quick schema changes and resets, facilitating agile development cycles. For production environments, transitioning to a more robust database system, such as PostgreSQL or MySQL, is recommended.

5.2 Frontend Implementation

The frontend is developed as a **Single Page Application (SPA)** using **React** and **Vite**.

Architecture

- **Component-Based Structure:** The UI is divided into reusable components, enhancing maintainability and scalability.
- **State Management:** Utilizes **React Context** and custom hooks to manage global state and side effects, ensuring a responsive user experience.
- **API Integration:** Axios is used for making HTTP requests to the backend. Interceptors are implemented to handle JWT tokens for authentication seamlessly.
- **Styling:** **Tailwind CSS** is employed for styling, providing a utility-first approach that speeds up the development process and ensures a consistent design.



Features

- **Responsive Design:** The application is optimized for various devices, including desktops, tablets, and smartphones.
- **Real-Time Filtering:** Users can filter job listings based on selected skills, locations, and other criteria, with results updating dynamically.
- **Interactive UI:** Features such as modals, tooltips, and dynamic forms enhance user engagement and experience.

5.3 Security Implementation

Security is a paramount concern in the system.

- **JWT Authentication:** Upon successful login, a JWT is issued and stored in the client's local storage. This token is included in the Authorization header of subsequent API requests to authenticate the user.
- **Role-Based Access Control (RBAC):** Different user roles (e.g., Admin, User) are defined, with specific permissions assigned to each role. This ensures that users can only access resources and perform actions that align with their roles.
- **CORS Configuration:** Cross-Origin Resource Sharing (CORS) policies are configured to restrict access to the API from unauthorized domains, enhancing security.
- **Data Validation:** Input data is validated both on the client-side and server-side to prevent malicious data from entering the system.

5.4 Containerization with Docker

To ensure consistency across development and production environments, the application is containerized using **Docker**.

Backend

- A **Dockerfile** is created to define the environment for the backend application. It specifies the base image, dependencies, and build instructions.
- The application is packaged into a **JAR file** and included in the Docker image, ensuring that all necessary components are bundled together.

Frontend

- The frontend application is built using **Vite**, and the production-ready files are served using **Nginx** within the Docker container.
- A separate **Dockerfile** is created for the frontend, defining its build and runtime environment.

Orchestration

- **Docker Compose** is used to define and run multi-container Docker applications. It allows for the configuration of both backend and frontend services, along with their dependencies, in a single YAML file.
- The docker-compose.yml file specifies the services, networks, and volumes required for the application, facilitating easy setup and teardown of the development environment.

TESTING AND RESULTS

6.1 Unit Testing

Unit tests are written using **JUnit 5** to validate the functionality of individual components.

- **Mockito** is used to mock dependencies, allowing for isolated testing of each unit.
- Test coverage is maintained at over **90%**, ensuring that the majority of the codebase is tested.
- Tests are organized to cover various scenarios, including edge cases and error conditions.

6.2 Integration Testing

Integration tests are conducted to verify that different components of the system work together as expected.

- **RestAssured** is utilized to perform HTTP requests and validate responses, ensuring that the API endpoints function correctly.



- Tests cover the complete flow, from user authentication to job posting and skill matching, to ensure end-to-end functionality.

6.3 Performance Testing

Performance testing is carried out to assess the system's responsiveness and stability under load.

- **Apache JMeter** is used to simulate multiple concurrent users performing various actions, such as logging in, searching for jobs, and applying for positions.
- Metrics such as response time, throughput, and resource utilization are monitored to identify potential bottlenecks.
- The system demonstrates an average response time of under **300ms** even under heavy load, indicating efficient performance.

6.4 Usability Testing

Usability testing is performed to evaluate the user experience of the application.

- A group of **20 participants** is selected to interact with the application and provide feedback.
- The **System Usability Scale (SUS)** is employed to quantify the usability of the system, with the application achieving a score of **88 out of 100**, indicating a high level of user satisfaction.
- Feedback highlights the intuitive navigation, responsive design, and effective filtering features as key strengths of the application.

CONCLUSION

The Optimized Job Matching System successfully addresses the limitations of traditional job portals by implementing a skill-based recommendation engine. Key achievements include:

- **Skill Taxonomy:** A comprehensive skill hierarchy enables nuanced matching between candidates and job postings.
- **Modular Architecture:** The system's design allows for easy maintenance and future enhancements.
- **Security:** Robust authentication and authorization mechanisms ensure data protection.
- **Performance:** The application performs efficiently under load, providing a seamless user experience.
- **Usability:** Positive feedback from users indicates high satisfaction with the system's functionality and interface.

Future Enhancements Could Include:

- **Machine Learning Integration:** Implementing machine learning algorithms to further refine job recommendations based on user behavior and preferences.
- **Persistent Database:** Transitioning to a production-grade database to support large-scale data storage and complex queries.
- **Resume Parsing:** Integrating resume parsing capabilities to automatically extract skills and qualifications from user-uploaded documents.
- **Mobile Application:** Developing a mobile version of the application to reach a broader audience and provide users with greater accessibility.

REFERENCES

- [1]. Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures (PhD thesis).
- [2]. Pivotal Software. (2024). Spring Boot Reference Documentation. <https://spring.io/projects/spring-boot>
- [3]. Facebook. (2024). React – A JavaScript library for building user interfaces. <https://reactjs.org/>
- [4]. Evan You. (2024). Vite. <https://vitejs.dev/>
- [5]. Jones, M. (2023). JWT.IO – JSON Web Tokens introduction. <https://jwt.io/>
- [6]. H2 Database Group. (2024). H2 Database Engine. <https://www.h2database.com/>
- [7]. Adam Wathan et al. (2024). Tailwind CSS Documentation. <https://tailwindcss.com/>
- [8]. Merkel, D. (2014). Docker: lightweight Linux containers for consistent development and deployment. Linux Journal.
- [9]. Apache JMeter. (2024). <https://jmeter.apache.org/>
- [10]. Postman, Inc. (2024). Postman Documentation. <https://www.postman.com/>